JACKSON - QUICK GUIDE

JACKSON - OVERVIEW

Jackson is a simple Java-based library to serialize Java objects to JSON and vice versa.

Features of Jackson

- **Easy to use** − Jackson API provides a high-level facade to simplify commonly used use-cases.
- **No need to create mapping** − Jackson API provides default mapping for most of the objects to be serialized.
- **Performance** − Jackson is quite fast, consumes less memory space, and is suitable for large object graphs or systems.
- **Clean JSON** − Jackson creates clean and compact JSON results which are easy to read.
- **No Dependency** − Jackson library does not require any other library apart from JDK.
- **Open Source** − Jackson library is open source and free to use.

Process JSON using Jackson

Jackson provides three different ways to process JSON −

- **Streaming API** − It reads and writes JSON content as discrete events. JsonParser reads the data, whereas JsonGenerator writes the data.
    - It is the most powerful approach among the three.
    - It has the lowest overhead and it provides the fastest way to perform read/write operations.
    - It is analogous to **Stax parser** for XML.
- **Tree Model** − It prepares an in-memory tree representation of the JSON document. ObjectMapper build tree of JsonNode nodes. It is most flexible approach. It is analogous to DOM parser for XML.
- **Data Binding** − It converts JSON to and from Plain Old Java Object (POJO) using property accessor or using annotations. ObjectMapper reads/writes JSON for both types of data bindings. Data binding is analogous to **JAXB parser** for XML. Data binding is of two types −
    - **Simple Data Binding** − It converts JSON to and from Java Maps, Lists, Strings, Numbers, Booleans, and null objects.
    - **Full Data Binding** − It converts JSON to and from any Java type.

JACKSON - ENVIRONMENT SETUP

> *This chapter describes how to set up the Jackson environment on your system.*
>
> ### *Try-It Online Option*
>
> *You really do not need to set up your own environment to start learning Jackson. We have set up an online Java Programming environment online, so that you can compile and execute all the available examples online. Feel free to modify any example and check the result with different options.*

> *Try the following example using the **Try it** option available at the top right corner of the sample code box on our website −*
>
> ```
> public class MyFirstJavaProgram {
>    public static void main(String []args) {
>       System.out.println("Hello World");
>    }
> }
> ```
>
> *For most of the examples given in this tutorial, you will find a **Try it** option to help you learn quickly through practice.*

Local Environment Setup

If you still wish to set up your environment for Java programming language, then this section guides you on how to download and set up Java on your machine. Follow the given steps to set up the environment.

Java SE is freely available from the link Download Java. You can download a version based on your operating system.

Follow the instructions to download Java and run the **.exe** to install Java on your machine. Once you have installed Java, you would need to set the environment variables to point to the correct installation directories.

Setting up the Path for Windows 2000/XP

Assuming you have installed Java in **c:\Program Files\java\jdk directory,**

- Right-click on 'My Computer'.
- Select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.
- Alter the 'Path' variable so that it also contains the path to the Java executable. For example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting up the Path for Windows 95/98/ME

Assuming you have installed Java in **c:\Program Files\java\jdk directory,**

- Edit the 'C:\autoexec.bat' file
- Add the following line at the end −

  'SET PATH=%PATH%;C:\Program Files\java\jdk\bin'

Setting up the Path for Linux, UNIX, Solaris, and FreeBSD

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

For example, if you use *bash* as your shell, then you need to add the following line at the end of your '.bashrc: export PATH=/path/to/java:$PATH'.

Popular Java Editors

To write Java programs, you need a text editor. There are sophisticated IDEs available in the market to write Java programs, but for now, you can consider one of the following −

- **Notepad** − On Windows platform, you can use any simple text editor such as Notepad (recommended for this tutorial) or TextPad.

- **Netbeans** − It is an open-source Java IDE. It can be downloaded from http://www.netbeans.org/index.html.
- **Eclipse** − It is also a Java IDE developed by the Eclipse open-source community. It can be downloaded from http://www.eclipse.org/.

Download Jackson Archive

Download the latest version of Jackson jar file from jackson-all-1.9.0.jar.zip. In this tutorial, *jackson-1.9.0.jar* is downloaded and copied into C:\>jackson folder.

OS	Archive name
Windows	jackson-all-1.9.0.jar
Linux	jackson-all-1.9.0.jar
Mac	jackson-all-1.9.0.jar

Set up Jackson Environment

Set the **jackson_HOME** environment variable to point to the base directory location where Jackson jar is stored on your machine. Depending on the platform you are working on, the process varies as shown in the following table −

OS	Output
Windows	Set the environment variable jackson_HOME to C:\jackson
Linux	export jackson_HOME=/usr/local/jackson
Mac	export jackson_HOME=/Library/jackson

Set CLASSPATH Variable

Set the **CLASSPATH** environment variable to point to the Jackson jar location.

OS	Output
Windows	Set the environment variable CLASSPATH to %CLASSPATH%;%jackson_HOME%\jackson-all-1.9.0.jar;.;
Linux	export CLASSPATH=$CLASSPATH:$jackson_HOME/jackson-all- 1.9.0.jar:.
Mac	export CLASSPATH=$CLASSPATH:$jackson_HOME/jackson-all- 1.9.0.jar:.

JACKSON - FIRST APPLICATION

Before going into the details of the Jackson library, let us see an application in action.

Jackson Example

In the following example, we will create a Student class. Thereafter, we will create a JSON string with Student details and deserialize it to Student object and then serialize it back to a JSON string.

Create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.IOException;

import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.map.SerializationConfig;

public class JacksonTester {
   public static void main(String args[]){
      ObjectMapper mapper = new ObjectMapper();
      String jsonString = "{\"name\":\"Mahesh\", \"age\":21}";
      //map json to student

      try {
         Student student = mapper.readValue(jsonString, Student.class);
         System.out.println(student);
         mapper.enable(SerializationConfig.Feature.INDENT_OUTPUT);
         jsonString = mapper.writeValueAsString(student);
         System.out.println(jsonString);
      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}

class Student {
   private String name;
   private int age;

   public Student(){}

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }
   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}
```

Verifythe Result

Compile the classes using **javac** compiler as follows −

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Execute the jacksonTester to see the result.

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output −

```
Student [ name: Mahesh, age: 21 ]
{
   "name" : "Mahesh",
   "age" : 21
}
```

Steps to Remember

Following are the important steps to be considered here.

Step 1: Create ObjectMapper Object

Create ObjectMapper object. It is a reusable object.

```
ObjectMapper mapper = new ObjectMapper();
```

Step 2: Deserialize JSON to Object

Use readValue() method to get the Object from the JSON. Pass the JSON string or the source of the JSON string and the object type as parameters.

```
//Object to JSON Conversion
Student student = mapper.readValue(jsonString, Student.class);
```

Step 3: Serialize Object to JSON

Use writeValueAsString() method to get the JSON string representation of an object.

```
//Object to JSON Conversion
jsonString = mapper.writeValueAsString(student);
```

JACKSON - OBJECTMAPPER CLASS

ObjectMapper is the main actor class of Jackson library. ObjectMapper class provides functionalities to convert Java objects to matching JSON constructs and vice versa. It uses instances of JsonParser and JsonGenerator for implementing actual reading/writing of JSON.

ObjectMapper Class Declaration

Following is the declaration for **org.codehaus.jackson.map.ObjectMapper** class −

```
public class ObjectMapper
   extends ObjectCodec
      implements Versioned
```

ObjectMapper Nested Classes

S.No.	Class & Description
1	**static class ObjectMapper.DefaultTypeResolverBuilder** Customized TypeResolverBuilder that provides type resolver builders used with so-called "default typing" (see enableDefaultTyping() for details).
2	**static class ObjectMapper.DefaultTyping** Enumeration used with enableDefaultTyping() to specify what kind of types (classes) default typing should be used for.

ObjectMapper Class Constructors

S.No.	Constructor & Description
1	**ObjectMapper()** It is the default constructor, which constructs the default JsonFactory as necessary. Use StdSerializerProvider as its SerializerProvider, and BeanSerializerFactory as its SerializerFactory.
2	**ObjectMapper(JsonFactory jf)** Construct mapper that uses specified JsonFactory for constructing necessary JsonParsers and/or JsonGenerators.
3	**ObjectMapper(JsonFactory jf, SerializerProvider sp, DeserializerProvider dp)**
4	**ObjectMapper(JsonFactory jf, SerializerProvider sp, DeserializerProvider dp, SerializationConfig sconfig, DeserializationConfig dconfig)**
5	**ObjectMapper(SerializerFactory sf)** Deprecated. Use other constructors instead; note that you can set the serializer factory with **setSerializerFactory(org.codehaus.jackson.map.SerializerFactory)**

ObjectMapper Methods

S.No.	Method & Description
1	**protected void _configAndWriteValue(JsonGenerator jgen, Object value)** Method called to configure the generator as necessary and then call write functionality.
2	**protected void _configAndWriteValue(JsonGenerator jgen, Object value, Class<?> viewClass)**
3	**protected Object _convert(Object fromValue, JavaType toValueType)**
4	**protected DeserializationContext _createDeserializationContext(JsonParser jp, DeserializationConfig cfg)**
5	**protected PrettyPrinter _defaultPrettyPrinter()** Helper method that should return default pretty-printer to use for generators constructed by this mapper, when instructed to use default pretty printer.
6	**protected JsonDeserializer<Object> _findRootDeserializer(DeserializationConfig cfg, JavaType valueType)** Method called to locate deserializer for the passed root-level value.
7	**protected JsonToken _initForReading(JsonParser jp)** Method called to ensure that given parser is ready for reading content for data binding.

8 **protected Object _readMapAndClose(JsonParser jp, JavaType valueType)**

9 **protected Object _readValue(DeserializationConfig cfg, JsonParser jp, JavaType valueType)**

Actual implementation of value reading + binding operation.

10 **protected Object _unwrapAndDeserialize(JsonParser jp, JavaType rootType, DeserializationContext ctxt, JsonDeserializer<Object> deser)**

11 **boolean canDeserialize(JavaType type)**

Method that can be called to check whether mapper thinks it could deserialize an Object of given type.

12 **boolean canSerialize(Class<?> type)**

Method that can be called to check whether mapper thinks it could serialize an instance of given Class.

13 **ObjectMapper configure(DeserializationConfig.Feature f, boolean state)**

Method for changing state of an on/off deserialization feature for this object mapper.

14 **ObjectMapper configure(JsonGenerator.Feature f, boolean state)**

Method for changing state of an on/off JsonGenerator feature for JsonFactory instance this object mapper uses.

15 **ObjectMapper configure(JsonParser.Feature f, boolean state)**

Method for changing state of an on/off JsonParser feature for JsonFactory instance this object mapper uses.

16 **ObjectMapper configure(SerializationConfig.Feature f, boolean state)**

Method for changing state of an on/off serialization feature for this object mapper.

17 **JavaType constructType(Type t)**

Convenience method for constructing JavaType out of given type (typically java.lang.Class), but without explicit context.

18 **<T> T convertValue(Object fromValue, Class<T> toValueType)**

Convenience method for doing two-step conversion from given value, into instance of given value type.

19 **<T> T convertValue(Object fromValue, JavaType toValueType)**

20 **<T> T convertValue(Object fromValue, TypeReference toValueTypeRef)**

21 **DeserializationConfig copyDeserializationConfig()**

Method that creates a copy of the shared default DeserializationConfig object that defines configuration settings for deserialization.

22 **SerializationConfig copySerializationConfig()**

Method that creates a copy of the shared default SerializationConfig object that defines configuration settings for serialization.

23 **ArrayNode createArrayNode()**

Note − return type is co-variant, as basic ObjectCodec abstraction cannot refer to concrete node types (as it is a part of core package, whereas impls are part of mapper package).

24 **ObjectNode createObjectNode()**

Note − return type is co-variant, as basic ObjectCodec abstraction cannot refer to concrete node types (as it is part of core package, whereas impls are part of mapper package)

25 **ObjectWriter defaultPrettyPrintingWriter()**

Deprecated. Since 1.9, use writerWithDefaultPrettyPrinter() instead.

26 **ObjectMapper disable(DeserializationConfig.Feature... f)**

Method for enabling specified DeserializationConfig features.

27 **ObjectMapper disable(SerializationConfig.Feature... f)**

Method for enabling specified DeserializationConfig features.

28 **ObjectMapper disableDefaultTyping()**

Method for disabling automatic inclusion of type information; if so, only explicitly annotated types (ones with JsonTypeInfo) has additional embedded type information.

29 **ObjectMapper enable(DeserializationConfig.Feature... f)**

Method for enabling specified DeserializationConfig features.

30 **ObjectMapper enable(SerializationConfig.Feature... f)**

Method for enabling specified DeserializationConfig features.

31 **ObjectMapper enableDefaultTyping()**

Convenience method that is equivalent to calling.

32 **ObjectMapper enableDefaultTyping(ObjectMapper.DefaultTyping dti)**

Convenience method that is equivalent to calling.

33 **ObjectMapper enableDefaultTyping(ObjectMapper.DefaultTyping applicability, JsonTypeInfo.As includeAs)**

Method for enabling automatic inclusion of type information, needed for proper deserialization of polymorphic types (unless types are annotated with JsonTypeInfo).

34 **ObjectMapper enableDefaultTypingAsProperty(ObjectMapper.DefaultTyping applicability, String propertyName)**

Method for enabling automatic inclusion of type information -- needed for proper deserialization of polymorphic types (unless types have been annotated with JsonTypeInfo) -- using "As.PROPERTY" inclusion mechanism and specified property name to use for inclusion (default being "@class" since default type information always uses class name as type identifier)

35 **ObjectWriter filteredWriter(FilterProvider filterProvider)**

Deprecated. Since 1.9, use writer(FilterProvider) instead.

36 **JsonSchema generateJsonSchema(Class<?> t)**

Generate Json-schema instance for specified class.

37 **JsonSchema generateJsonSchema(Class<?> t, SerializationConfig cfg)**

Generate Json-schema instance for specified class, using specific serialization configuration

38 **DeserializationConfig getDeserializationConfig()**

Method that returns the shared default DeserializationConfig object that defines configuration settings for deserialization.

39 **DeserializerProvider getDeserializerProvider()**

40 **JsonFactory getJsonFactory()**

Method that can be used to get hold of JsonFactory that this mapper uses if it needs to construct JsonParsers and/or JsonGenerators.

41 **JsonNodeFactory getNodeFactory()**

Method that can be used to get hold of JsonNodeFactory that this mapper will use when directly constructing root JsonNode instances for Trees.

42 **SerializationConfig getSerializationConfig()**

Method that returns the shared default SerializationConfig object that defines configuration settings for serialization.

43 **SerializerProvider getSerializerProvider()**

44 **SubtypeResolver getSubtypeResolver()**

Method for accessing subtype resolver in use.

45 **TypeFactory getTypeFactory()**

Accessor for getting currently configured TypeFactory instance.

46 **VisibilityChecker<?> getVisibilityChecker()**

Method for accessing currently configured visibility checker; object used for determining whether given property element (method, field, constructor) can be auto-detected or not.

47 **boolean isEnabled(DeserializationConfig.Feature f)**

48 **boolean isEnabled(JsonGenerator.Feature f)**

49 **boolean isEnabled(JsonParser.Feature f)**

50 **boolean isEnabled(SerializationConfig.Feature f)**

51 **ObjectWriter prettyPrintingWriter(PrettyPrinter pp)**

52 **ObjectReader reader()**

Factory method for constructing ObjectReader with default settings.

53 **ObjectReader reader(Class<?> type)**

Factory method for constructing ObjectReader that reads or update instances of specified type.

54 **ObjectReader reader(FormatSchema schema)**

Factory method for constructing ObjectReader that will pass specific schema object to JsonParser used for reading content.

55 **ObjectReader reader(InjectableValues injectableValues)**

Factory method for constructing ObjectReader that will use specified injectable values.

56 **ObjectReader reader(JavaType type)**

Factory method for constructing ObjectReader that will read or update instances of specified type.

57 **ObjectReader reader(JsonNodeFactory f)**

Factory method for constructing ObjectReader that will use specified JsonNodeFactory for constructing JSON trees.

58 **ObjectReader reader(TypeReference<?> type)**

Factory method for constructing ObjectReader that will read or update instances of specified type.

59 **ObjectReader readerForUpdating(Object valueToUpdate)**

Factory method for constructing ObjectReader that will update given Object (usually Bean, but can be a Collection or Map as well, but NOT an array) with JSON data.

60 **JsonNode readTree(byte[] content)**

Method to deserialize JSON content as tree expressed using set of JsonNode instances.

61 **JsonNode readTree(File file)**

Method to deserialize JSON content as tree expressed using set of JsonNode instances.

62 **JsonNode readTree(InputStream in)**

Method to deserialize JSON content as tree expressed using set of JsonNode instances.

63 **JsonNode readTree(JsonParser jp)**

Method to deserialize JSON content as tree expressed using set of JsonNode instances.

64 **JsonNode readTree(JsonParser jp, DeserializationConfig cfg)**

Method to deserialize JSON content as tree expressed using set of JsonNode instances.

65 **JsonNode readTree(Reader r)**

Method to deserialize JSON content as tree expressed using set of JsonNode instances.

66 **JsonNode readTree(String content)**

Method to deserialize JSON content as tree expressed using set of JsonNode instances.

67 **JsonNode readTree(URL source)**

Method to deserialize JSON content as tree expressed using set of JsonNode instances.

68 **<T> T readValue(byte[] src, Class<T> valueType)**

69 **<T> T readValue(byte[] src, int offset, int len, Class<T> valueType)**

70 **<T>** T readValue(byte[] src, int offset, int len, JavaType valueType)

71 **<T> T readValue(byte[] src, int offset, int len, TypeReference valueTypeRef)**

72 **<T> T readValue(byte[] src, JavaType valueType)**

73 **<T> T readValue(byte[] src, TypeReference valueTypeRef)**

74 **<T> T readValue(File src, Class<T> valueType)**

75 **<T> T readValue(File src, JavaType valueType)**

76 **<T> T readValue(File src, TypeReference valueTypeRef)**

77 **<T> T readValue(InputStream src, Class<T> valueType)**

78 **<T> T readValue(InputStream src, JavaType valueType)**

79 **<T> T readValue(InputStream src, TypeReference valueTypeRef)**

80 **<T> T readValue(JsonNode root, Class<T> valueType)**

Convenience method for converting results from given JSON tree into given value type.

81 **<T> T readValue(JsonNode root, JavaType valueType)**

Convenience method for converting results from given JSON tree into given value type.

82 **<T> T readValue(JsonNode root, TypeReference valueTypeRef)**

Convenience method for converting results from given JSON tree into given value type.

83 **<T> T readValue(JsonParser jp, Class<T> valueType)**

Method to deserialize JSON content into a non-container type (it can be an array type, however) − typically a bean, array or a wrapper type (like Boolean).

84 **<T> T readValue(JsonParser jp, Class<T> valueType, DeserializationConfig cfg)**

Method to deserialize JSON content into a non-container type (it can be an array type, however) − typically a bean, array or a wrapper type (like Boolean).

85 **<T> T readValue(JsonParser jp, JavaType valueType)**

Method to deserialize JSON content into a Java type, reference to which is passed as argument.

86 **<T> T readValue(JsonParser jp, JavaType valueType, DeserializationConfig cfg)**

Method to deserialize JSON content into a Java type, reference to which is passed as argument.

87 **<T> T readValue(JsonParser jp, TypeReference<?> valueTypeRef)**

Method to deserialize JSON content into a Java type, reference to which is passed as argument.

88 **<T> T readValue(JsonParser jp, TypeReference<?> valueTypeRef, DeserializationConfig cfg)**

Method to deserialize JSON content into a Java type, reference to which is passed as argument.

89 **<T> T readValue(Reader src, Class<T> valueType)**

90 **<T> T readValue(Reader src, JavaType valueType)**

91 **<T> T readValue(Reader src, TypeReference valueTypeRef)**

92 **<T> T readValue(String content, Class<T> valueType)**

93 **<T> T readValue(String content, JavaType valueType)**

94 **<T> T readValue(String content, TypeReference valueTypeRef)**

95 **<T> T readValue(URL src, Class<T> valueType)**

96 **<T> T readValue(URL src, JavaType valueType)**

97 **<T> T readValue(URL src, TypeReference valueTypeRef)**

98 **<T> MappingIterator<T> readValues(JsonParser jp, Class<T> valueType)**

Method for reading sequence of Objects from parser stream.

99 **<T> MappingIterator<T> readValues(JsonParser jp, JavaType valueType)**

Method for reading sequence of Objects from parser stream.

100 **<T> MappingIterator<T> readValues(JsonParser jp, TypeReference<?> valueTypeRef)**

Method for reading sequence of Objects from parser stream.

101 **void registerModule(Module module)**

Method for registering a module that can extend functionality provided by this mapper; for example, by adding providers for custom serializers and deserializers.

102 **void registerSubtypes(Class<?>... classes)**

Method for registering specified class as a subtype, so that typename-based resolution can link supertypes to subtypes (as an alternative to using annotations).

103 **void registerSubtypes(NamedType... types)**

Method for registering specified class as a subtype, so that typename-based resolution can link supertypes to subtypes (as an alternative to using annotations).

104 **ObjectReader schemaBasedReader(FormatSchema schema)**

105 **ObjectWriter schemaBasedWriter(FormatSchema schema)**

106 **ObjectMapper setAnnotationIntrospector(AnnotationIntrospector ai)**

Method for changing AnnotationIntrospector used by this mapper instance for both serialization and deserialization.

107 **void setDateFormat(DateFormat dateFormat)**

Method for configuring the default DateFormat to use when serializing time values as Strings, and deserializing from JSON Strings.

108 **ObjectMapper setDefaultTyping(TypeResolverBuilder<?> typer)**

Method for enabling automatic inclusion of type information, using specified handler object for determining which types this affects, as well as details of how information is embedded.

109 **ObjectMapper setDeserializationConfig(DeserializationConfig cfg)**

Method for replacing the shared default deserialization configuration object.

110 **ObjectMapper setDeserializerProvider(DeserializerProvider p)**

Method for setting specific DeserializerProvider to use for handling the caching of JsonDeserializer instances.

111 **void setFilters(FilterProvider filterProvider)**

112 **void setHandlerInstantiator(HandlerInstantiator hi)**

Method for configuring HandlerInstantiator to use for creating instances of handlers (such as serializers, deserializers, type and type id resolvers), given a class.

113 **ObjectMapper setInjectableValues(InjectableValues injectableValues)**

114 **ObjectMapper setNodeFactory(JsonNodeFactory f)**

Method for specifying JsonNodeFactory to use for constructing root level tree nodes (via method createObjectNode()

115 **ObjectMapper setPropertyNamingStrategy(PropertyNamingStrategy s)**

Method for setting custom property naming strategy to use.

116 **ObjectMapper setSerializationConfig(SerializationConfig cfg)**

Method for replacing the shared default serialization configuration object.

117 **ObjectMapper setSerializationInclusion(JsonSerialize.Inclusion incl)**

Method for setting default POJO property inclusion strategy for serialization.

118 **ObjectMapper setSerializerFactory(SerializerFactory f)**

Method for setting specific SerializerFactory to use for constructing (bean) serializers.

119 **ObjectMapper setSerializerProvider(SerializerProvider p)**

Method for setting specific SerializerProvider to use for handling caching of JsonSerializer instances.

120 **void setSubtypeResolver(SubtypeResolver r)**

Method for setting custom subtype resolver to use.

121 **ObjectMapper setTypeFactory(TypeFactory f)**

Method that can be used to override TypeFactory instance used by this mapper.

122 **ObjectMapper setVisibility(JsonMethod forMethod, JsonAutoDetect.Visibility visibility)**

Convenience method that allows changing configuration for underlying VisibilityCheckers, to change details of what kinds of properties are auto-detected.

123 **void setVisibilityChecker(VisibilityChecker<?> vc)**

Method for setting currently configured visibility checker; object used to determine whether given property element (method, field, constructor) can be auto-detected or not.

125 **JsonParser treeAsTokens(JsonNode n)**

Method for constructing a JsonParser out of JSON tree representation.

126 **<T> T treeToValue(JsonNode n, Class<T> valueType)**

Convenience conversion method that binds data provided the JSON tree contains a specific value (usually bean) type.

127 **ObjectWriter typedWriter(Class<?> rootType)**

Deprecated. Since 1.9, use writerWithType(Class) instead.

128 **ObjectWriter typedWriter(JavaType rootType)**

Deprecated. Since 1.9, use writerWithType(JavaType) instead.

129 **ObjectWriter typedWriter(TypeReference<?> rootType)**

Deprecated. Since 1.9, use writerWithType(TypeReference) instead.

130 **ObjectReader updatingReader(Object valueToUpdate)**

Deprecated. Since 1.9, use readerForUpdating(java.lang.Object) instead.

131 **<T extends JsonNode> T valueToTree(Object fromValue)**

Reverse of treeToValue(org.codehaus.jackson.JsonNode, java.lang.Class); given a value (usually bean), constructs equivalent JSON Tree representation.

132 **Version version()**

Method that will return version information stored and reads from jar that contains this class.

133 **ObjectWriter viewWriter(Class<?> serializationView)**

Deprecated. Since 1.9, use writerWithView(Class) instead.

134 **ObjectMapper withModule(Module module) Fluent-style alternative to registerModule(org.codehaus.jackson.map.Module);**

135 **ObjectWriter writer()**

Convenience method for constructing ObjectWriter with default settings.

136 **ObjectWriter writer(DateFormat df)**

Factory method for constructing ObjectWriter that serializes objects using specified DateFormat; or, if null passed, using timestamp (64-bit number.

137 **ObjectWriter writer(FilterProvider filterProvider)**

Factory method for constructing ObjectWriter that serializes objects using specified filter provider.

138 **ObjectWriter writer(FormatSchema schema)**

Factory method for constructing ObjectWriter that passes specific schema object to JsonGenerator used for writing content.

139 **ObjectWriter writer(PrettyPrinter pp)**

Factory method for constructing ObjectWriter that serializes objects using specified pretty printer for indentation (or if null, no pretty printer)

140 **ObjectWriter writerWithDefaultPrettyPrinter()**

Factory method for constructing ObjectWriter that serializes objects using the default pretty printer for indentation.

141 **ObjectWriter writerWithType(Class<?> rootType)**

Factory method for constructing ObjectWriter that serializes objects using specified root type, instead of actual runtime type of value.

142 **ObjectWriter writerWithType(JavaType rootType)**

Factory method for constructing ObjectWriter that serializes objects using specified root

type, instead of actual runtime type of value.

143 **ObjectWriter writerWithType(TypeReference<?> rootType)**

Factory method for constructing ObjectWriter that serializes objects using specified root type, instead of actual runtime type of value.

144 **ObjectWriter writerWithView(Class<?> serializationView)**

Factory method for constructing ObjectWriter that serializes objects using specified JSON View (filter).

145 **void writeTree(JsonGenerator jgen, JsonNode rootNode)**

Method to serialize given JSON Tree, using generator provided.

146 **void writeTree(JsonGenerator jgen, JsonNode rootNode, SerializationConfig cfg)**

Method to serialize given Json Tree, using generator provided.

147 **void writeValue(File resultFile, Object value)**

Method that can be used to serialize any Java value as JSON output, written to File provided.

148 **void writeValue(JsonGenerator jgen, Object value)**

Method that can be used to serialize any Java value as JSON output, using provided JsonGenerator.

149 **void writeValue(JsonGenerator jgen, Object value, SerializationConfig config)**

Method that can be used to serialize any Java value as JSON output, using provided JsonGenerator, configured as per passed configuration object.

150 **void writeValue(OutputStream out, Object value)**

Method that can be used to serialize any Java value as JSON output, using output stream provided (using encoding JsonEncoding.UTF8).

151 **void writeValue(Writer w, Object value)**

Method that can be used to serialize any Java value as JSON output, using Writer provided.

152 **byte[] writeValueAsBytes(Object value)**

Method that used to serialize any Java value as a byte array.

153 **String writeValueAsString(Object value)**

Method that can be used to serialize any Java value as a String.

ObjectMapper class inherits methods from java.lang.Object.

ObjectMapper Example

Create the following Java program using any editor of your choice and save it in the folder **C:/> Jackson_WORKSPACE**

File: JacksonTester.java

```
import java.io.IOException;

import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.map.SerializationConfig;

public class JacksonTester {
   public static void main(String args[]){
      ObjectMapper mapper = new ObjectMapper();
      String jsonString = "{\"name\":\"Mahesh\", \"age\":21}";

      //map json to student
      try {
         Student student = mapper.readValue(jsonString, Student.class);
         System.out.println(student);
         mapper.enable(SerializationConfig.Feature.INDENT_OUTPUT);
         jsonString = mapper.writeValueAsString(student);
         System.out.println(jsonString);

      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}

class Student {
   private String name;
   private int age;

   public Student(){}

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }

   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}
```

Verify the Result

Compile the classes using **javac** compiler as follows −

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now execute the jacksonTester to see the result.

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the output –

```
Student [ name: Mahesh, age: 21 ]
{
   "name" : "Mahesh",
   "age" : 21
}
```

JACKSON - OBJECT SERIALIZATION

To understand object serialization in detail, let us serialize a Java object to a JSON file and then read that JSON file to get the object back.

Object Serialization Example

In the following example, we will create a Student class. Thereafter we will create a student.json file which will have a JSON representation of Student object.

First of all, create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;

import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();

      try {
         Student student = new Student();
         student.setAge(10);
         student.setName("Mahesh");
         tester.writeJSON(student);

         Student student1 = tester.readJSON();
         System.out.println(student1);

      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

   private void writeJSON(Student student) throws JsonGenerationException,
JsonMappingException, IOException{
      ObjectMapper mapper = new ObjectMapper();
      mapper.writeValue(new File("student.json"), student);
   }

   private Student readJSON() throws JsonParseException, JsonMappingException,
```

```
IOException{
      ObjectMapper mapper = new ObjectMapper();
      Student student = mapper.readValue(new File("student.json"), Student.class);
      return student;
   }
}

class Student {
   private String name;
   private int age;

   public Student(){}

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }

   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows −

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result.

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output −

```
Student [ name: Mahesh, age: 10 ]
```

JACKSON - DATA BINDING

Data Binding API is used to convert JSON to and from Plain Old Java Object (POJO) using property accessor or using annotations. It is of two types −

- **Simple Data Binding** − It converts JSON to and from Java Maps, Lists, Strings, Numbers, Booleans, and null objects.
- **Full Data Binding** − It converts JSON to and from any Java type.

ObjectMapper reads/writes JSON for both types of data bindings. Data binding is analogous to JAXB parser for XML.

We will cover simple data binding in this chapter. Full data binding is discussed separately in the next chapter.

Simple Data Binding

Simple data binding refers to mapping of JSON to JAVA core data types. The following table illustrates the relationship between JSON types versus Java types.

S.No.	JSON Type	Java Type
1	object	LinkedHashMap<String, Object>
2	array	ArrayList<Object>
3	string	String
4	complete number	Integer, Long or BigInteger
5	fractional number	Double / BigDecimal
6	true | false	Boolean
7	null	null

Simple Data Binding Example

Let us take a simple example to understand simple data binding in detail. Here, we'll map Java basic types directly to JSON and vice versa.

Create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;

import java.util.HashMap;
import java.util.Map;

import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();

      try {
         ObjectMapper mapper = new ObjectMapper();

         Map<String,Object> studentDataMap = new HashMap<String,Object>();
         int[] marks = {1,2,3};

         Student student = new Student();

         student.setAge(10);
         student.setName("Mahesh");

         // JAVA Object
         studentDataMap.put("student", student);

         // JAVA String
         studentDataMap.put("name", "Mahesh Kumar");

         // JAVA Boolean
         studentDataMap.put("verified", Boolean.FALSE);

         // Array
         studentDataMap.put("marks", marks);
```

```
            mapper.writeValue(new File("student.json"), studentDataMap);

            //result student.json
            //{
               //   "student":{"name":"Mahesh","age":10},
               //   "marks":[1,2,3],
               //   "verified":false,
               //   "name":"Mahesh Kumar"
            //}

            studentDataMap = mapper.readValue(new File("student.json"), Map.class);

            System.out.println(studentDataMap.get("student"));
            System.out.println(studentDataMap.get("name"));
            System.out.println(studentDataMap.get("verified"));
            System.out.println(studentDataMap.get("marks"));

         } catch (JsonParseException e) {
            e.printStackTrace();
         } catch (JsonMappingException e) {
            e.printStackTrace();
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
   }

   class Student {
      private String name;
      private int age;

      public Student(){}

      public String getName() {
         return name;
      }

      public void setName(String name) {
         this.name = name;
      }

      public int getAge() {
         return age;
      }

      public void setAge(int age) {
         this.age = age;
      }

      public String toString(){
         return "Student [ name: "+name+", age: "+ age+ " ]";
      }
   }
```

Verify the Result

Compile the classes using **javac** compiler as follows –

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result –

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output –

```
{name=Mahesh, age=10}
```

```
Mahesh Kumar
false
[1, 2, 3]
```

JACKSON - FULL DATA BINDING

Full data binding refers to mapping of JSON to any Java Object.

```
//Create an ObjectMapper instance
ObjectMapper mapper = new ObjectMapper();

//map JSON content to Student object
Student student = mapper.readValue(new File("student.json"), Student.class);

//map Student object to JSON content
mapper.writeValue(new File("student.json"), student);
```

Full Data Binding Example

Let us take a simple example to understand full data binding in detail. In the following example, we will map a Java Object directly to JSON and vice versa.

Create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;

import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();

      try {
         Student student = new Student();
         student.setAge(10);
         student.setName("Mahesh");
         tester.writeJSON(student);

         Student student1 = tester.readJSON();
         System.out.println(student1);

      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

   private void writeJSON(Student student) throws JsonGenerationException,
JsonMappingException, IOException{
      ObjectMapper mapper = new ObjectMapper();
      mapper.writeValue(new File("student.json"), student);
   }

   private Student readJSON() throws JsonParseException, JsonMappingException,
IOException{
      ObjectMapper mapper = new ObjectMapper();
      Student student = mapper.readValue(new File("student.json"), Student.class);
      return student;
```

```
   }
}

class Student {
   private String name;
   private int age;

   public Student(){}

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }

   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}
```

Verify the Result

Compile the classes using **javac** compiler as follows −

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now execute the jacksonTester to see the result.

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output −

```
Student [ name: Mahesh, age: 10 ]
```

JACKSON - DATA BINDING WITH GENERICS

In simple data binding, we have used Map class which uses String as key and Object as a value object. Instead, we can have a concrete Java object and type cast it to use it in JSON binding.

Consider the following example with a class UserData, a class to hold user-specific data.

Create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;

import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
```

```
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.type.TypeReference;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         ObjectMapper mapper = new ObjectMapper();

         Map<String, UserData> userDataMap = new HashMap<String, UserData>();
         UserData studentData = new UserData();
         int[] marks = {1,2,3};

         Student student = new Student();
         student.setAge(10);
         student.setName("Mahesh");

         // JAVA Object
         studentData.setStudent(student);

         // JAVA String
         studentData.setName("Mahesh Kumar");

         // JAVA Boolean
         studentData.setVerified(Boolean.FALSE);

         // Array
         studentData.setMarks(marks);
         TypeReference ref = new TypeReference<Map<String,UserData>>() { };
         userDataMap.put("studentData1", studentData);
         mapper.writeValue(new File("student.json"), userDataMap);

         //{

         //   "studentData1":
         //   {
         //      "student":

         //   {
         //      "name":"Mahesh",
         //      "age":10
         //   },

         //   "name":"Mahesh Kumar",
         //   "verified":false,
         //   "marks":[1,2,3]
         //   }

         //}
         userDataMap = mapper.readValue(new File("student.json"), ref);

         System.out.println(userDataMap.get("studentData1").getStudent());
         System.out.println(userDataMap.get("studentData1").getName());
         System.out.println(userDataMap.get("studentData1").getVerified());
         System.out.println(Arrays.toString(userDataMap.get("studentData1").getMarks()));

      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}

class Student {
   private String name;
   private int age;
```

```
   public Student(){}

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }

   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}

class UserData {
   private Student student;
   private String name;
   private Boolean verified;
   private int[] marks;

   public UserData(){}

   public Student getStudent() {
      return student;
   }

   public void setStudent(Student student) {
      this.student = student;
   }

   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }

   public Boolean getVerified() {
      return verified;
   }

   public void setVerified(Boolean verified) {
      this.verified = verified;
   }
   public int[] getMarks() {
      return marks;
   }

   public void setMarks(int[] marks) {
      this.marks = marks;
   }
}
```

Verify the Result

Compile the classes using **javac** compiler as follows –

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result –

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output –

```
Student [ name: Mahesh, age: 10 ]
Mahesh Kumar
false
[1, 2, 3]
```

JACKSON - TREE MODEL

The Tree Model prepares an in-memory tree representation of a JSON document. It is the most flexible approach among the three processing modes that Jackson supports. It is quite similar to DOM parser in XML.

Create a Tree from JSON

ObjectMapper provides a pointer to root node of the tree after reading the JSON. Root Node can be used to traverse the complete tree. Consider the following code snippet to get the root node of a provided JSON String.

```
//Create an ObjectMapper instance
ObjectMapper mapper = new ObjectMapper();
String jsonString = "{\"name\":\"Mahesh Kumar\",
\"age\":21,\"verified\":false,\"marks\": [100,90,85]}";

//create tree from JSON
JsonNode rootNode = mapper.readTree(jsonString);
```

Traversing a Tree

Get each node using the relative path to the root node while traversing the tree and process the data. The following code snippet shows how to traverse a tree, provided you have information regarding the root node.

```
JsonNode nameNode = rootNode.path("name");
System.out.println("Name: "+ nameNode.getTextValue());

JsonNode marksNode = rootNode.path("marks");
Iterator iterator = marksNode.getElements();
```

Tree Model Example

Create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.IOException;
import java.util.Iterator;

import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         ObjectMapper mapper = new ObjectMapper();
         String jsonString = "{\"name\":\"Mahesh Kumar\",
\"age\":21,\"verified\":false,\"marks\": [100,90,85]}";
```

```
         JsonNode rootNode = mapper.readTree(jsonString);

         JsonNode nameNode = rootNode.path("name");
         System.out.println("Name: "+ nameNode.getTextValue());

         JsonNode ageNode = rootNode.path("age");
         System.out.println("Age: " + ageNode.getIntValue());

         JsonNode verifiedNode = rootNode.path("verified");
         System.out.println("Verified: " + (verifiedNode.getBooleanValue() ?
"Yes":"No"));

         JsonNode marksNode = rootNode.path("marks");
         Iterator<JsonNode> iterator = marksNode.getElements();
         System.out.print("Marks: [ ");

         while (iterator.hasNext()) {
            JsonNode marks = iterator.next();
            System.out.print(marks.getIntValue() + " ");
         }

         System.out.println("]");
      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Verify the Result

Compile the classes using **javac** compiler as follows −

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now execute the jacksonTester to see the result.

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output −

```
Name: Mahesh Kumar
Age: 21
Verified: No
Marks: [ 100 90 85 ]
```

Tree to JSON Conversion

In the following example, we will create a Tree using JsonNode and write it to a JOSN file and read it back.

Create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;

import java.util.Iterator;

import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.JsonParseException;
```

```
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.node.ArrayNode;
import org.codehaus.jackson.node.ObjectNode;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         ObjectMapper mapper = new ObjectMapper();

         JsonNode rootNode = mapper.createObjectNode();
         JsonNode marksNode = mapper.createArrayNode();
         ((ArrayNode)marksNode).add(100);
         ((ArrayNode)marksNode).add(90);
         ((ArrayNode)marksNode).add(85);
         ((ObjectNode) rootNode).put("name", "Mahesh Kumar");
         ((ObjectNode) rootNode).put("age", 21);
         ((ObjectNode) rootNode).put("verified", false);
         ((ObjectNode) rootNode).put("marks",marksNode);

         mapper.writeValue(new File("student.json"), rootNode);

         rootNode = mapper.readTree(new File("student.json"));

         JsonNode nameNode = rootNode.path("name");
         System.out.println("Name: "+ nameNode.getTextValue());

         JsonNode ageNode = rootNode.path("age");
         System.out.println("Age: " + ageNode.getIntValue());

         JsonNode verifiedNode = rootNode.path("verified");
         System.out.println("Verified: " + (verifiedNode.getBooleanValue() ?
"Yes":"No"));

         JsonNode marksNode1 = rootNode.path("marks");
         Iterator<JsonNode> iterator = marksNode1.getElements();
         System.out.print("Marks: [ ");

         while (iterator.hasNext()) {
            JsonNode marks = iterator.next();
            System.out.print(marks.getIntValue() + " ");
         }

         System.out.println("]");
      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Verify the Result

Compile the classes using **javac** compiler as follows –

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result.

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output –

```
Name: Mahesh Kumar
Age: 21
Verified: No
Marks: [ 100 90 85 ]
```

Tree to Java Objects

In the following example, we will perform the following operations −

- Create a Tree using JsonNode
- Write it to a JSON file
- Read back the tree and then convert it to a Student object.

First of all, create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;

import java.util.Arrays;

import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.node.ArrayNode;
import org.codehaus.jackson.node.ObjectNode;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         ObjectMapper mapper = new ObjectMapper();

         JsonNode rootNode = mapper.createObjectNode();
         JsonNode marksNode = mapper.createArrayNode();
         ((ArrayNode)marksNode).add(100);
         ((ArrayNode)marksNode).add(90);
         ((ArrayNode)marksNode).add(85);
         ((ObjectNode) rootNode).put("name", "Mahesh Kumar");
         ((ObjectNode) rootNode).put("age", 21);
         ((ObjectNode) rootNode).put("verified", false);
         ((ObjectNode) rootNode).put("marks",marksNode);

         mapper.writeValue(new File("student.json"), rootNode);

         rootNode = mapper.readTree(new File("student.json"));

         Student student = mapper.treeToValue(rootNode, Student.class);

         System.out.println("Name: "+ student.getName());
         System.out.println("Age: " + student.getAge());
         System.out.println("Verified: " + (student.isVerified() ? "Yes":"No"));
         System.out.println("Marks: "+Arrays.toString(student.getMarks()));

      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

```
class Student {
   String name;
   int age;
   boolean verified;
   int[] marks;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }

   public boolean isVerified() {
      return verified;
   }

   public void setVerified(boolean verified) {
      this.verified = verified;
   }

   public int[] getMarks() {
      return marks;
   }

   public void setMarks(int[] marks) {
      this.marks = marks;
   }
}
```

Verify the Result

Compile the classes using **javac** compiler as follows −

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now execute the jacksonTester to see the result.

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output −

```
Name: Mahesh Kumar
Age: 21
Verified: No
Marks: [ 100 90 85 ]
```

JACKSON - STREAMING API

Streaming API reads and writes JSON content as discrete events. JsonParser reads the data, whereas JsonGenerator writes the data.

- It is the most powerful approach among the three processing modes that Jackson supports.
- It has the lowest overhead and it provides the fastest way to perform read/write operations.
- It is analogous to Stax parser for XML.

In this chapter, we will discuss how to read and write JSON data using Jackson streaming APIs. Streaming API works with the concept of token and every details of JSON is to be handled carefully. Following are the two classes which we will use in the examples given in this chapter −

- **JsonGenerator** − Write to JSON String.
- **JsonParser** − Parse JSON String.

Write to JSON using JsonGenerator

It is pretty simple to use JsonGenerator. First, create the JsonGenerator using JsonFactory.createJsonGenerator() method and use its write***() methods to write each JSON value.

```
JsonFactory jasonFactory = new JsonFactory();
JsonGenerator jsonGenerator = jasonFactory.createJsonGenerator(new File("student.json"),
JsonEncoding.UTF8);

// {
   jsonGenerator.writeStartObject();
// "name" : "Mahesh Kumar"

jsonGenerator.writeStringField("name", "Mahesh Kumar");
```

Let us see JsonGenerator in action. Create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;

import java.util.Map;

import org.codehaus.jackson.JsonEncoding;
import org.codehaus.jackson.JsonFactory;
import org.codehaus.jackson.JsonGenerator;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();

      try {
         JsonFactory jasonFactory = new JsonFactory();

         JsonGenerator jsonGenerator = jasonFactory.createJsonGenerator(new
File("student.json"), JsonEncoding.UTF8);

         // {
            jsonGenerator.writeStartObject();

            // "name" : "Mahesh Kumar"
            jsonGenerator.writeStringField("name", "Mahesh Kumar");

            // "age" : 21
            jsonGenerator.writeNumberField("age", 21);

            // "verified" : false
            jsonGenerator.writeBooleanField("verified", false);

            // "marks" : [100, 90, 85]
            jsonGenerator.writeFieldName("marks");

            // [
               jsonGenerator.writeStartArray();
```

```
               // 100, 90, 85
               jsonGenerator.writeNumber(100);
               jsonGenerator.writeNumber(90);
               jsonGenerator.writeNumber(85);
            // ]

            jsonGenerator.writeEndArray();

         // }

         jsonGenerator.writeEndObject();
         jsonGenerator.close();

         //result student.json

         //{
            //   "name":"Mahesh Kumar",
            //   "age":21,
            //   "verified":false,
            //   "marks":[100,90,85]
         //}

         ObjectMapper mapper = new ObjectMapper();
         Map<String,Object> dataMap = mapper.readValue(new File("student.json"),
Map.class);

         System.out.println(dataMap.get("name"));
         System.out.println(dataMap.get("age"));
         System.out.println(dataMap.get("verified"));
         System.out.println(dataMap.get("marks"));

      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Verify the Result

Compile the classes using **javac** compiler as follows −

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now execute the jacksonTester to see the result.

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the output −

```
Mahesh Kumar
21
false
[100, 90, 85]
```

Reading JSON using JsonParser

Using JsonParser is again pretty simple. First, create the JsonParser using JsonFactory.createJsonParser() method and use its nextToken() method to read each JSON string as token. Check each token and process accordingly.

```
JsonFactory jasonFactory = new JsonFactory();
JJsonParser jsonParser = jasonFactory.createJsonParser(new File("student.json"));
```

```
while (jsonParser.nextToken() != JsonToken.END_OBJECT) {
   //get the current token
   String fieldname = jsonParser.getCurrentName();

   if ("name".equals(fieldname)) {
      //move to next token
      jsonParser.nextToken();
      System.out.println(jsonParser.getText());
   }
}
```

Let us see JsonParser in action. Create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;

import org.codehaus.jackson.JsonEncoding;
import org.codehaus.jackson.JsonFactory;
import org.codehaus.jackson.JsonGenerator;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.JsonParser;
import org.codehaus.jackson.JsonToken;
import org.codehaus.jackson.map.JsonMappingException;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         JsonFactory jasonFactory = new JsonFactory();

         JsonGenerator jsonGenerator = jasonFactory.createJsonGenerator(new
File("student.json"), JsonEncoding.UTF8);
         jsonGenerator.writeStartObject();
         jsonGenerator.writeStringField("name", "Mahesh Kumar");
         jsonGenerator.writeNumberField("age", 21);

         jsonGenerator.writeBooleanField("verified", false);
         jsonGenerator.writeFieldName("marks");

         jsonGenerator.writeStartArray(); // [

         jsonGenerator.writeNumber(100);
         jsonGenerator.writeNumber(90);
         jsonGenerator.writeNumber(85);

         jsonGenerator.writeEndArray();
         jsonGenerator.writeEndObject();

         jsonGenerator.close();

         //result student.json

         //{
            //   "name":"Mahesh Kumar",
            //   "age":21,
            //   "verified":false,
            //   "marks":[100,90,85]
         //}

         JsonParser jsonParser = jasonFactory.createJsonParser(new File("student.json"));

         while (jsonParser.nextToken() != JsonToken.END_OBJECT) {
            //get the current token
            String fieldname = jsonParser.getCurrentName();
```

```
            if ("name".equals(fieldname)) {
               //move to next token
               jsonParser.nextToken();
               System.out.println(jsonParser.getText());
            }

            if("age".equals(fieldname)){
               //move to next token
               jsonParser.nextToken();
               System.out.println(jsonParser.getNumberValue());
            }

            if("verified".equals(fieldname)){
               //move to next token
               jsonParser.nextToken();
               System.out.println(jsonParser.getBooleanValue());
            }

            if("marks".equals(fieldname)){
               //move to [
               jsonParser.nextToken();
               // loop till token equal to "]"

               while (jsonParser.nextToken() != JsonToken.END_ARRAY) {
                  System.out.println(jsonParser.getNumberValue());
               }
            }
         }
      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Verify the Result

Compile the classes using **javac** compiler as follows −

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result.

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output −

```
Mahesh Kumar
21
false
[100, 90, 85]
```